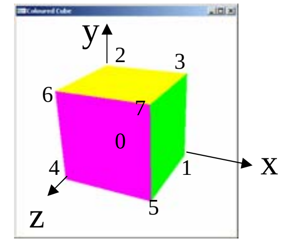
7. Modelling with Polygon Meshes

So far, we have dealt only with wireframe 3D models. Solid objects can be modelled by polygons representing their surface.

7.1 ColourCube Example




```
// A program to display a simple cube with each face a different colour.
// Cube can be rotated with a trackball.
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
#include "Trackball.h"

const int windowWidth=400;
const int windowHeight=400;

// define vertices and faces of the cube
const int numVertices=8;
const int numFaces=6;
const int numFaceVertices=4;
const float vertices[numVertices][3] = {{0,0,0},{1,0,0},{0,1,0},{1,1,0},{0,0,1},{1,0,1},{0,1,1},{1,1,1}};
const int faces[numFaces][numFaceVertices] = { {0,1,5,4}, {1,3,7,5}, // Bottom, Right
                                               {0,4,6,2}, {2,6,7,3}, // Left, Top
                                               {4,5,7,6}, {0,2,3,1}}; // Front, Back
const float faceColours[numFaces][3] =         // (R,G,B) colours of each face
    {{1,0,0},{0,1,0},{0,0,1},{1,1,0},{1,0,1},{0,1,1}}; // red, green, blue, yellow, magenta, cyan
CTrackball trackball;
```


ColourCube Example (cont'd)

```
void handleMouseMotion(int x, int y) { trackball.tbMotion(x, y); }
void handleMouseClick(int button, int state, int x, int y) {trackball.tbMouse(button, state, x, y); }
void handleKeyboardEvent(unsigned char key, int x, int y) { trackball.tbKeyboard(key); }

void display(void){
    glMatrixMode( GL_MODELVIEW );  // Set the view matrix ...
    glLoadIdentity();              // ... to identity.
    gluLookAt(0,0,4, 0,0,0, 0,1,0); // camera is on the z-axis
    trackball.tbMatrix();          // rotate the cube using the trackball ...
    glTranslatef(-0.5f,-0.5f,-0.5f); // ... and move it to the centre

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear colour buffer
                                                        // and depth buffer
    for (int i=0; i < numFaces; i++) {
        const int* face = faces[i];
        glBegin(GL_POLYGON);
        glColor3fv(faceColours[i]);
        for (int vIndex=0; vIndex<numFaceVertices; vIndex++)
                glVertex3fv(vertices[face[vIndex]]);
        glEnd();}
    glFlush ();
    glutSwapBuffers();             // swap framebuffer in which image has been draw with
}                                  // the frame buffer read by the CRT controller
```


ColourCube Example (cont'd)

```
void init(void)
{
    // select clearing color (for glClear)
    glClearColor (1,1,1,1);    // RGB-value for white
    // enable depth buffering
    glEnable(GL_DEPTH_TEST);
    // initialize view (simple orthographic projection)
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(33,1,2,8);
    trackball.tbInit(GLUT_LEFT_BUTTON);
}

void reshape(int width, int height ) {
    // Called at start, and whenever user resizes component
    int size = min(width, height);
    glViewport(0, 0, size, size);        // Largest possible square
    trackball.tbReshape(width, height);
}
```

ColourCube Example (cont'd)

```
// create a double buffered colour window
int main(int argc, char** argv)
{
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
     glutInitWindowSize(windowWidth, windowHeight);
     glutInitWindowPosition(100, 100);
     glutCreateWindow("Coloured Cube");
     init ();                                // initialise view
     glutMouseFunc(handleMouseClick);        // Set function to handle mouse clicks
     glutMotionFunc(handleMouseMotion);      // Set function to handle mouse motion
     glutKeyboardFunc(handleKeyboardEvent);  // Set function to handle keyboard input
     glutDisplayFunc(display);               // Set function to draw scene
     glutReshapeFunc(reshape);               // Set function called if window gets resized
     glutMainLoop();
     return 0;
}
```


Notes on the ColourCube Example

- Polyhedra are solid objects with polygonal faces
  - Usually represented as an array of vertices and an array of polygonal faces, each face being an array of indices into the vertex array.
- IMPORTANT: the vertices of a face must be stored in ANTICLOCKWISE order when looking from outside
  - i.e. right-handed with respect to outgoing face normal
  - Ordering is used by OpenGL to classify faces as *front* or *back* w.r.t. eye point.
  - Wrongly classified faces may be coloured wrongly (see later)
- Use *double buffering* for animations :
  - In `main`: glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  - In `display`: glutSwapBuffers();
    - As you rotate the cube, the new image is rendered into an off-screen buffer, then buffers are swapped.
    - Without these lines, animation can look messy

Notes on the ColourCube Example (cont'd)

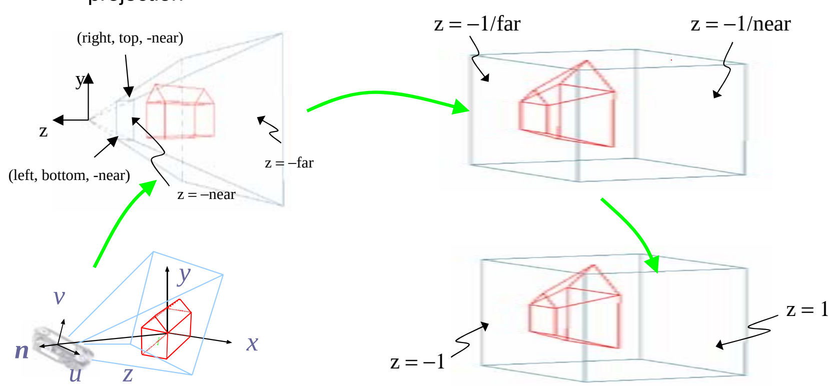
- Enable *depth testing* as polygons are drawn to the screen
  `glEnable(GL_DEPTH_TEST);`
  - Only front faces are visible [see next slide]
- `display` method does a *perspective* projection with the view point being on the z-axis (more in chapter 8 of this course)
- Before drawing scene, you must clear the frame buffer (aka colour buffer) AND the depth buffer with:
  `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);`
- The output primitives are polygons
  - Between `glBegin(GL_POLYGON)` and `glEnd()` you output all the vertices of the (convex) polygon
- Polygon is drawn with whatever the current colour is.
  - Note: OpenGL can't fill concave polygons
  - You have to break them into convex bits

7.2 The Depth Buffer

- When polygons overlap, we want to see only the nearest one
  - "Visible surface determination" aka "Hidden surface removal"
- Achieved by using a *depth buffer*
  - projection defines for each pixel of the colour buffer a depth value which corresponds to the normalised distance of that pixel to the view point (camera):
    depthBuffer[i][j] = $-z_{PerpectiveSpace}$ of pixel stored at colourBuffer[i][j]
- As each new polygon is drawn, its depth is computed at each pixel
  - Its colour and depth are copied into the colourBuffer and depthBuffer only if its depth is less than the current depth
  - glEnable(GL_DEPTH_TEST) turns on this feature
  - Depth buffer must be cleared (to value 1.0f) at the same time as colour buffer is cleared

The Depth Buffer (cont'd)

- Depth test performed after view transformation and perspective projection

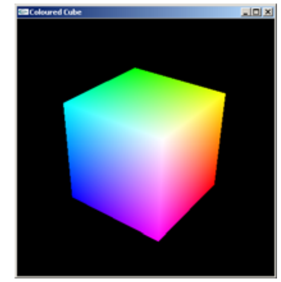


Perform depth test in these coordinates!!

7.3 RGBColourCube Example

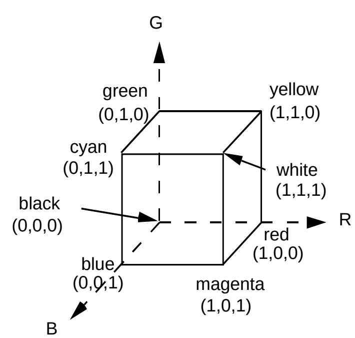
- Rather than having each face a fixed colour, we can set each vertex to a different colour.
- If we set the (R,G,B) colour of each vertex of the unit cube to equal its spatial coordinates, we draw the "RGB Colour Cube"
  - Main output loop in `display` becomes:

```
for (int i=0; i < numFaces; i++) {
    const int* face = faces[i];
    glBegin(GL_POLYGON);
    for (int vIndex=0; vIndex<numFaceVertices; vIndex++)
    {
      glColor3fv(vertices[face[vIndex]]);
      glVertex3fv(vertices[face[vIndex]]);
    }
    glEnd();
}
```



RGBColourCube Example (cont'd)

- Computer screens have a fine dot pattern of red, green and blue phosphor dots
  - A separate electron gun acitvates each colour
- Hence all computer colours are generated by additive mixing of red, green and blue.
- So a solid RGB cube contains all colours that can be produced on a monitor
  - But it's a subset of the space of all VISIBLE colours

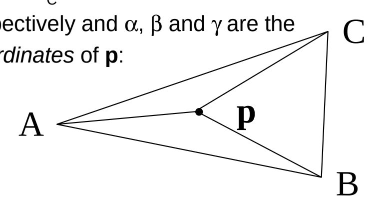


RGBColourCube Example (cont'd)

- OpenGL renders polygons by breaking them into triangles
- If each triangle vertex has a colour the colours are interpolated over the polygon: Within a triangle ABC, the colour at any point $\mathbf{p} = \alpha A + \beta B + \gamma C$

is $C_p = \alpha C_A + \beta C_B + \gamma C_C$

where C_A, C_B, and C_C are the colours at the vertices A, B and C respectively and α, β and γ are the *barycentric coordinates* of **p**:



$$\alpha = \frac{area(\Delta_{\mathbf{p}BC})}{area(\Delta_{ABC})}$$

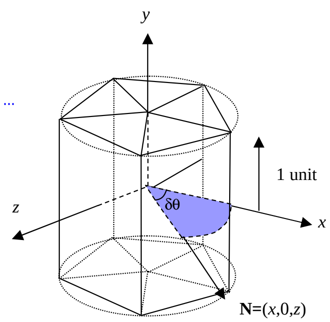
$$\beta = \frac{area(\Delta_{\mathbf{p}CA})}{area(\Delta_{ABC})}$$

$$\gamma = \frac{area(\Delta_{\mathbf{p}AB})}{area(\Delta_{ABC})}$$

7.4 Shading 3D Objects

Up to now we created coloured surfaces without considering light sources. Example: A coloured cylinder

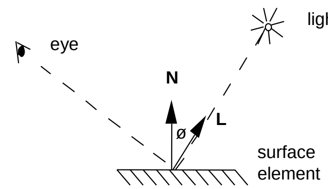
```
void display(void)
{
    glMatrixMode( GL_MODELVIEW ); // Set the view matrix ...
    glLoadIdentity();             // ... to identity.
    gluLookAt(0,0,6, 0,0,0, 0,1,0); // camera is on the z-axis
    trackball.tbMatrix();         // rotate the cylinder using the trackball ...
    glColor3f(1,0,0);             // cylinder is red

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    float x,z,theta;
    glBegin(GL_QUAD_STRIP);
    for (int segment=0; segment <= NUM_SEGMENTS; segment++){
        theta=2.0f*Pi*(float) segment/(float) NUM_SEGMENTS;
        x = (float) cos(theta);   // x = r cos(theta), and r = 1
        z = (float) sin(theta);   // z = r sin(theta)
        glVertex3f(x,-1, z);
        glVertex3f(x,1,z);
    }
    glEnd();
    glFlush (); glutSwapBuffers();
}
```

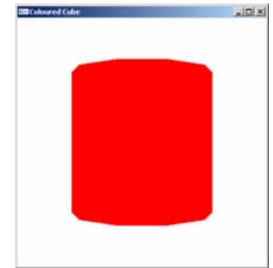


Note: Figure shows an example with 5 segment.

A Coloured Cylinder (cont'd)

- The resulting image is very hard to interpret since all faces have the same colour.



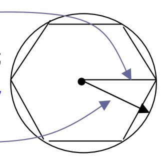


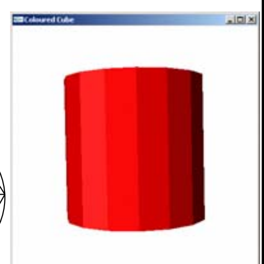
- In natural scenes objects are illuminated. The perceived colour of a polygon is influenced by its material properties, its orientation with respect to the light source and the view point, and the light's colour.
- The orientation of a surface is defined by specifying a surface normal for each polygon (*flat shading*) or for each vertex (*smooth shading*).
- We will do more on illumination and shading later in this lecture

A Flat-Shaded Cylinder

- An improved image of the scene is obtained by defining material properties & light sources and one normal for each polygon:
  - All points of a polygon will have the same perceived colour, but different polygons might have different perceived colours according the orientation of the polygon normal with respect to the light source.

```
glShadeModel(GL_FLAT);
...
glBegin(GL_QUAD_STRIP);
for (int segment=0; segment <= NUM_SEGMENTS; segment++)
{
    theta=2.0f*Pi*(float) segment/(float) NUM_SEGMENTS;
    x = (float) cos(theta);  // x = r cos(theta), and r = 1
    z = (float) sin(theta);  // z = r sin(theta)
    glVertex3f(x,-1, z);
    glVertex3f(x,1,z);
    theta=2.0f*Pi*((float) segment+0.5f)/(float) NUM_SEGMENTS;
    x = (float) cos(theta);
    z = (float) sin(theta);  // if defining individual polygons specify
    glNormal3f(x,0,z);       // one normal for each polygon
}
glEnd();
```

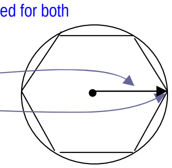


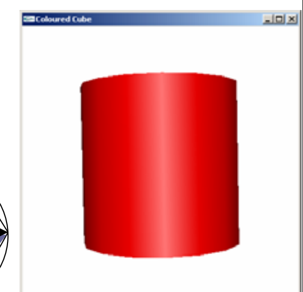


A Smooth-Shaded Cylinder

- In order implement smooth shading (Gouraud shading) define the true surface normal for each vertex of a polygon mesh:
  - OpenGL computes the perceived colour at each vertex of a polygon and then interpolates the colour (similar to slide 12).

```
glShadeModel(GL_SMOOTH);
...
glBegin(GL_QUAD_STRIP);
for (int segment=0; segment <= NUM_SEGMENTS; segment++)
{
    theta=2.0f*Pi*(float) segment/(float) NUM_SEGMENTS;
    x = (float) cos(theta);  // x = r cos(theta), and r = 1
    z = (float) sin(theta);  // z = r sin(theta)
    glNormal3f(x,0,z);       // Usually define one normal for each vertex. Here the
                             // Here the same normal is used for both
    glVertex3f(x,-1, z);     // the top vertex and
    glVertex3f(x,1,z);       // the bottom vertex
}
glEnd();
```



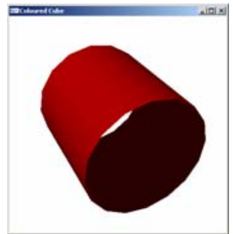


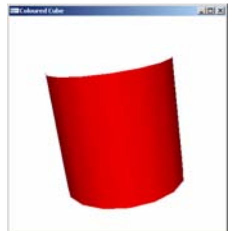
Rendering Polygon Meshes

- By default only one side of a polygon (the side on which the light source is located), is shaded.
  - No problem when modelling solid objects (such as a cube) since back faces are not visible.
  - Double-sided shading is possible in OpenGL.
- Since back faces of solid objects are not visible (back faces are always covered by front faces) rendering speed can be improved by eliminating these polygons:

```
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

(backfaces are identified by testing whether its anticlockwise ordered vertices are ordered clockwise when projected onto the screen)



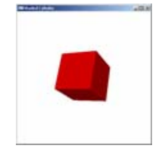


7.5 GLUT Functions

- The utility libraries GLUT provides a number of functions for rendering solid objects using polygon meshes:

  `glutSolidCube(GLdouble size)`
  - renders a solid cube centred at the origin with length `size`.
  - equivalent functions exist for drawing the corresponding objects as wire frames, eg. `glutWireCube(GLdouble size)`

  `glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks )`
  - renders a solid cone oriented along the z-axis with base at z=0, the top at z=height, and a base radius of `base`. The cone is subdivided around the z-axis into slices and along the z-axis into stacks.



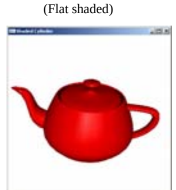
glutWireCone(1,2,8,2)

GLUT Functions (cont'd)

- `glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)`
  - renders a solid sphere centred at the origin with radius `radius`. The sphere is subdivided around the z-axis into slices and along the z-axis into stacks.

  `glutSolidTeapot(GLdouble size)`
  - renders a solid teapot centred at the origin with a diameter of approximately `2*size`.

glutSolidSphere(1,6,8)
(Flat shaded)

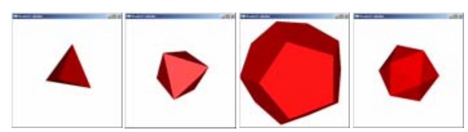


GLUT Functions (cont'd)

- `glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings)`
  - renders a solid torus (doughnut) centred at the origin whose axis is aligned with the z-axis. The torus is subdivided into `rings` segments along its circular centre line and into `nsides` segments around that line.

```
glutSolidTetrahedron()
glutSolidOctahedron()
glutSolidDodecahedron()
glutSolidIcosahedron()
```

glutSolidTorus(0.2, 1.0, 6, 8)
(Flat shaded)



Platonic Solids

7.6 Modelling using Matrix Operations

An object can be transformed using the OpenGL matrix operations

glTranslatef(GLfloat dx, GLfloat dy, GLfloat dz)
glScalef(GLfloat xFactor, GLfloat yFactor, GLfloat zFactor)
glRotatef(GLfloat angleInDegrees, GLfloat axisX, GLfloat axisY, GLfloat axisZ)
glMultMatrixf(const GLfloat *m)   // general purpose matrix

Many real-world objects can be modelled by combining and transforming more basic models. Example:

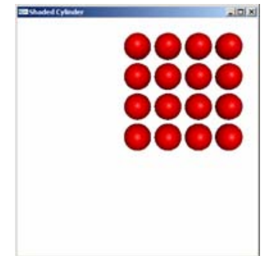
```
glutSolidTorus(0.1,1.0,24,32);
glRotatef(90,1,0,0);
glutSolidTorus(0.1,1.0,24,32);
glRotatef(90,0,1,0);
glutSolidTorus(0.1,1.0,24,32);
glutSolidSphere(0.4,32,32);
```

Modelling using Matrix Operations (cont'd)

- Problem: matrices put onto the matrix stack apply to **all** subsequently drawn objects.
- Often this is undesirable, e.g. imagine you want to draw a 4x4 matrix of spheres.

  A row in x-direction can be drawn by applying after each step a transformation in x-direction. Before drawing the next row a transformation must be applied in order to shift one level up in y-direction and back to the beginning of the row in x-direction.

```
float radius=0.2, shift=2*radius+0.05;
for(int j=0;j<4;j++){
    for(int i=0;i<4;i++){
          glutSolidSphere(radius,20,20);
          glTranslatef(shift,0,0);
    }
    glTranslatef(-4*shift,shift,0);
}
```



Modelling using Matrix Operations (cont'd)

- It is much more convenient to specify for each sphere its position, without having to worry about how the transformations influence subsequent drawing commands.
- This can be achieved by using `glPushMatrix()` and `glPopMatrix()`.
- In order to explain these commands we have to explain how a matrix stack (such as GL_MODEL_VIEW) works:
  - After initialisation with the identity matrix the matrix stack contains one element, ie. the identity matrix.
  - If an object is drawn then each point of the object (specified by `glVertex()`) is multiplied by the current top of the matrix stack.
  - If a transformation is applied (eg. `glTranslatef()`) then the current top of the matrix stack is multiplied on the right with the new matrix and the result replaces the matrix at the top of the stack.
  - `glPushMatrix()` makes a copy of the top of the matrix stack and pushes it on top of the stack. Any subsequent transformation matrices are therefore multiplied with that copy.
  - The (modified) copy is removed using `glPopMatrix()`. The new top of the matrix stack is the matrix on top before calling `glPushMatrix()`.

Modelling using Matrix Operations (cont'd)

- Here is an example demonstrating how various matrix operations change the GL_MODELVIEW matrix stack:

```
1. glMatrixMode( GL_MODELVIEW );
2. glLoadIdentity();                  // matrix I
3. gluLookAt(0,0,6, 0,0,0, 0,1,0);    // matrix V
4. trackball.tbMatrix();              // matrix B
5. glPushMatrix();
6. glTranslatef(tx,ty,tz);            // matrix T
7. glPopMatrix();
```

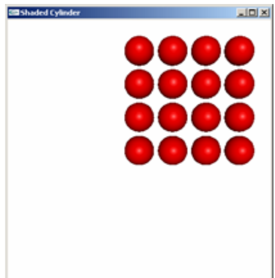
The GL_MODELVIEW matrix stack

				I*V*B	I*V*B*T	
	I	I*V	I*V*B	I*V*B	I*V*B	I*V*B
1.	2.	3.	4.	5.	6.	7.

Modelling using Matrix Operations (cont'd)

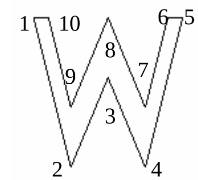
- We can now write rewrite our original program as follows:

```
float r=0.2, shift=2*r+0.05;
for(int j=0;j<4;j++)
  for(int i=0;i<4;i++){
    glPushMatrix();
    glTranslatef(i*shift,j*shift,0);
    glutSolidSphere(r,20,20);
    glPopMatrix();
  }
```

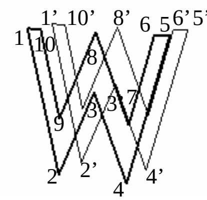


7.7 Extruded Surfaces

- Many 3D objects can be constructed by taking a 2D object and extruding it into a third dimension.
- The line strip defining the outline of the original 2D object becomes a quad strip defining the surface of the extruded object.
- Example:



Original 2D object



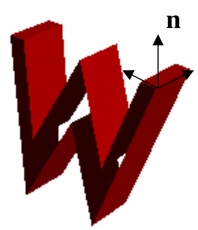
Original 2D object and the same object translated into a third dimension

Extruded surface represented by the quad strip 1,1',2,2',3,3',…

Add front and back faces (1,2,9,10; 2,3,8,9; …)

Extruded Surfaces (cont'd)

- The polygons representing the front and back faces can be specified "by hand".
- A more elegant solution is found, however, by noting that the front and the back face are concave (ie. non-convex) polygons. GLU provides a function to tessellate such polygons (ie. to subdivide them into triangles).
  - `gluNewTess()` returns such a polygon tessellator.
  - outside the scope of this lecture, but useful if you want to go into graphics :-)
- The surface normal of each polygon of the extruded surface is given by the cross product of each segment of the original line strip and the extrusion direction.



Extruded Surfaces (cont'd)

- Code for the extruded surface example:

```
const int numVertices=10;
const float vertices[numVertices][2] =
{{0,1},{0.25f,0},{0.5f,0.6f},{0.75f,0},{1,1},
  {0.9f,1},{0.75f,0.4f},{0.5f,1},{0.25f,0.4f},{0.1f,1}};

// in display()
glBegin(GL_QUAD_STRIP);
for(int i=0;i<=numVertices;i++){
    glVertex3f(vertices[i%10][0], vertices[i%10][1], 0.0);
    glVertex3f(vertices[i%10][0], vertices[i%10][1], -0.4);
    CVec3df v1(0,0,-0.4);
    CVec3df v2( vertices[(i+1)%10][0]-vertices[i%10][0],
                vertices[(i+1)%10][1]-vertices[i%10][1],0);
    CVec3df n=cross(v1,v2);
    n.normaliseDestructive();
    glNormal3fv(n.getArray());
}
glEnd();
```

Extruded Surfaces (cont'd)

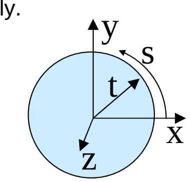
```
// front face
glBegin(GL_QUAD_STRIP);
glNormal3f(0,0,1.0);
glVertex3f(vertices[4][0], vertices[4][1], 0.0);
glVertex3f(vertices[5][0], vertices[5][1], 0.0);
glVertex3f(vertices[3][0], vertices[3][1], 0.0);
glVertex3f(vertices[6][0], vertices[6][1], 0.0);
glVertex3f(vertices[2][0], vertices[2][1], 0.0);
glVertex3f(vertices[7][0], vertices[7][1], 0.0);
glVertex3f(vertices[1][0], vertices[1][1], 0.0);
glVertex3f(vertices[8][0], vertices[8][1], 0.0);
glVertex3f(vertices[0][0], vertices[0][1], 0.0);
glVertex3f(vertices[9][0], vertices[9][1], 0.0);
glEnd();
```


7.8 Parametric Surfaces

- A parametric surface is defined as the set of points $\mathbf{p}(s,t)=\begin{pmatrix} x(s,t) \\ y(s,t) \\ z(s,t) \end{pmatrix}$

where *x(s,t)*, *y(s,t)*, and *z(s,t)* are functions of the parameters *s* and *t*, which lie within the intervals $[s_{min}, s_{max}]$ and $[t_{min}, t_{max}]$, respectively.

Example: $\mathbf{p}(s,t)=\begin{pmatrix} t\cos(2\pi s) \\ t\sin(2\pi s) \\ 0 \end{pmatrix}, \; t\in[0,1], s\in[0,1]$

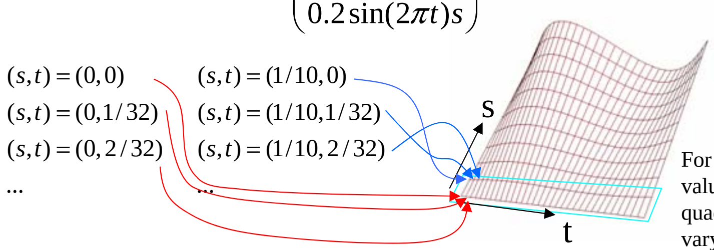


defines a disc with radius 1 and the z-axis as normals.

Parametric Surfaces (cont'd)

- In order to draw the surface subdivide the parameter intervals into equally sized steps, compute the vertex at each step, and connect the vertices to form quadrilaterals.

Example: $\mathbf{p}(s,t)=\begin{pmatrix} t \\ s \\ 0.2\sin(2\pi t)s \end{pmatrix}, \; t\in[0,1], s\in[0,1]$



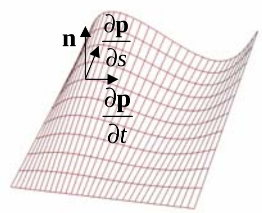
For each pair of s values draw one quad strip by varying *t*.

Parametric Surfaces (cont'd)

```
int numSegmentsS=10, numSegmentsT=32;
int i,j;
float s,t;
for(i=0;i<numSegmentsS;i++){
    glBegin(GL_QUAD_STRIP);
    for(j=0;j<=numSegmentsT;j++){
          s=(float) i/(float) numSegmentsS;
          t=(float) j/(float) numSegmentsT;
          glVertex3f(t,s,0.2*sin(2*Pi*t)*s);
          s=(float) (i+1)/(float) numSegmentsS;
          glVertex3f(t,s,0.2*sin(2*Pi*t)*s);
    }
    glEnd();
}
```

Parametric Surfaces (cont'd)

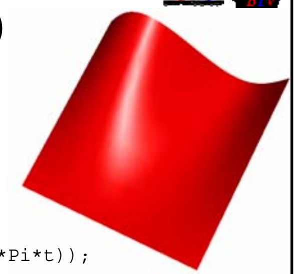
- Problem: What are the surface normals at each point?
  Idea: For each point $\mathbf{p}(s,t)$ compute the tangents of the surface in s and t direction. The surface normal $\mathbf{n}$ is perpendicular to both tangents.



$$\mathbf{p}(s,t)=\begin{pmatrix} t \\ s \\ 0.2\sin(2\pi t)s \end{pmatrix}$$

$$\frac{\partial \mathbf{p}}{\partial t}=\begin{pmatrix} 1 \\ 0 \\ 0.2\ 2\pi\cos(2\pi t)s \end{pmatrix},\ \frac{\partial \mathbf{p}}{\partial s}=\begin{pmatrix} 0 \\ 1 \\ 0.2\ \sin(2\pi t) \end{pmatrix},\ \mathbf{n}(s,t)=\frac{\partial \mathbf{p}}{\partial t}\times\frac{\partial \mathbf{p}}{\partial s}$$

Parametric Surfaces (cont'd)

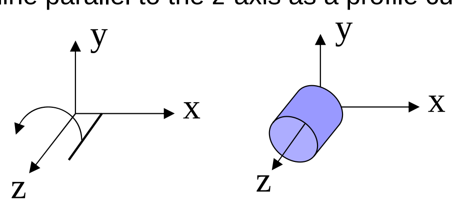


```
int i, j, numSegmentsS=10, numSegmentsT=32;
float s,t; CVec3df v1,v2,n;
for(i=0; i<numSegmentsS; i++){
    glBegin(GL_QUAD_STRIP);
    for(j=0; j<=numSegmentsT; j++){
            s=(float) i/(float) numSegmentsS;
            t=(float) j/(float) numSegmentsT;
            v1.setVector(1,0,0.2*s*2*Pi*cos(2*Pi*t));
            v2.setVector(0,1,0.2*sin(2*Pi*t));
            n=cross(v1,v2); n.normaliseDestructive();
            glNormal3fv(n.getArray());
            glVertex3f(t,s,0.2*sin(2*Pi*t)*s);
            s=(float) (i+1)/(float) numSegmentsS;
            v1.setVector(1,0,0.2*s*2*Pi*cos(2*Pi*t));
            v2.setVector(0,1,0.2*sin(2*Pi*t));
            n=cross(v1,v2); n.normaliseDestructive();
            glNormal3fv(n.getArray());
            glVertex3f(t,s,0.2*sin(2*Pi*t)*s);}
    glEnd();}
```


7.9 Surfaces of Revolution

- Surfaces of Revolution are extremely common in natural scenes (eg. pillars).
- They are formed by a *rotational sweep* of a *profile curve* around an axis.
- Without loss of generality we can assume that the profile curve is defined in the xz-plane and that it is rotated around the z-axis.

  Example: Using a line parallel to the z-axis as a profile curve gives a cylinder.



Surfaces of Revolution (cont'd)

- The profile curve is a parametric curve with the coordinates $(x(t), 0, z(t))$, $t\in[t_{min},t_{max}]$.
- If we rotate a point on this curve by an angle s around the z-axis we get the point $(x(t)\cos(s), x(t)\sin(s), z(t))$.
- A surface of revolution is therefore a parametric surface with the coordinates

$$\mathbf{p}(s,t)=\begin{pmatrix} x(t)\cos s \\ x(t)\sin s \\ z(t) \end{pmatrix},\ t\in[t_{min},t_{max}],\ s\in[0,2\pi]$$

Radius of the circular profile at z=z(t).

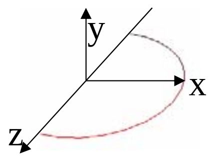
- The normal vector at a point (s,t) is $\mathbf{n}(s,t)=x(t)\begin{pmatrix} z'(t)\cos s \\ z'(t)\sin s \\ -x'(t) \end{pmatrix}$

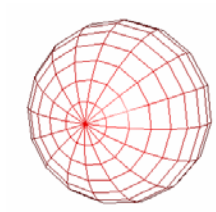
  where $z'(t)$ is the derivative of z.

Surfaces of Revolution (cont'd)

- Example: A sphere can be defined by taking a half circle as a profile curve.

$$\begin{pmatrix} x(t) \\ z(t) \end{pmatrix} = \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}, \ t \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$$





$$\mathbf{p}(s,t) = \begin{pmatrix} x(t)\cos s \\ x(t)\sin s \\ z(t) \end{pmatrix} = \begin{pmatrix} \cos t \cos s \\ \cos t \sin s \\ \sin t \end{pmatrix}, \ t \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right], s \in [0, 2\pi]$$

$$\mathbf{n}(s,t) = \mathbf{p}(s,t) \text{ [after normalisation]}$$

Surfaces of Revolution (cont'd)

```
int lt, lg, numLatitudes=16, numLongitudes=16;
double x,y,z,r;
for(lt=0; lt<numLatitudes-1; lt++){
    glBegin(GL_QUAD_STRIP);
    for(lg=0; lg<=numLongitudes; lg++){
            z = sin(Pi*lt/((double) (numLatitudes-1))-Pi/2.0f);
            r = cos(Pi*lt/((double) (numLatitudes-1))-Pi/2.0f);
            x = cos(2*Pi*lg/(double) numLongitudes)*r;
            y = sin(2*Pi*lg/(double) numLongitudes)*r;
            glNormal3f(x,y,z);
            glVertex3f(x,y,z);
            z = sin(Pi*(lt+1)/((double) (numLatitudes-1))-Pi/2.0f);
            r = cos(Pi*(lt+1)/((double) (numLatitudes-1))-Pi/2.0f);
            x = cos(2*Pi*lg/(double) numLongitudes)*r;
            y = sin(2*Pi*lg/(double) numLongitudes)*r;
            glNormal3f(x,y,z);
            glVertex3f(x,y,z);
    }
    glEnd();
}
```

NOTE: Faces are defined in clockwise direction. Use `glFrontFace(GL_CW);` if enabling back face culling.